软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(10):3293−3309 [doi: 10.13328/j.cnki.jos.006022]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

VMOffset:虚拟机自省中一种语义重构改进方法*

陈兴蜀[1,2], 蔡梦娟[1,2], 王 伟[2,3], 王启旭[1,2], 金 鑫[2,3]

[1](四川大学 网络空间安全学院,四川 成都 610207)

[2](四川大学 网络空间安全研究院,四川 成都 610207)

[3](四川大学 计算机学院,四川 成都 610065)

通讯作者: 陈兴蜀, E-mail: chenxsh@scu.edu.cn

摘 要: 虚拟机自省是一种在虚拟机外部获取目标虚拟机信息,并对其运行状态进行监控分析的方法.针对现有虚拟机自省方法在语义重构过程中存在的可移植性差、效率较低的问题,提出了一种语义重构改进方法 VMOffset.该方法基于进程结构体成员自身属性制定约束条件,可在不知道目标虚拟机内核版本的情况下,自动获取其进程结构体关键成员偏移量,所得偏移量可提供给开源或自主研发的虚拟机自省工具完成语义重构.在 KVM(kernel-based virtual machine)虚拟化平台上实现了 VMOffset 原型系统,并基于不同内核版本操作系统的虚拟机,对 VMOffset 的有效性及性能进行实验分析.结果表明:VMOffset 可自动完成各目标虚拟机中进程级语义的重构过程,具有可移植性与安全性,且仅对目标虚拟机的启动阶段引入 0.05%之内的性能损耗.

关键词: 虚拟机自省;语义重构;偏移量;虚拟机监视器;可移植性

中图法分类号: TP303

中文引用格式: 陈兴蜀,蔡梦娟,王伟,王启旭,金鑫.VMOffset:虚拟机自省中一种语义重构改进方法.软件学报,2021,32(10):3293−3309. http://www.jos.org.cn/1000-9825/6022.htm

英文引用格式: Chen XS, Cai MJ, Wang W, Wang QX, Jin X. VMOffset: Semantic reconstruction improvement method in virtual machine introspection. Ruan Jian Xue Bao/Journal of Software, 2021,32(10):3293−3309 (in Chinese). http://www.jos.org.cn/1000-9825/6022.htm

VMOffset: Semantic Reconstruction Improvement Method in Virtual Machine Introspection

CHEN Xing-Shu[1,2], CAI Meng-Juan[1,2], WANG Wei[2,3], WANG Qi-Xu[1,2], JIN Xin[2,3]

[1](School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China)

[2](Cyber Science Research Institute, Sichuan University, Chengdu 610207, China)

[3](College of Computer Science, Sichuan University, Chengdu 610065, China)

Abstract: Virtual machine introspection is a method to acquire the information of the target virtual machine, and monitor as well as analyze its running status outside the target virtual machine. Aiming at the problem of poor portability and low efficiency in the process of semantic reconstruction of existing virtual machine introspection method, a sematic reconstruction improvement method is proposed in this study. In this method, constraint conditions are made based on the characteristics of the process structure members, and the offsets of the process structure key members are automatically obtained without knowing the kernel version of the target virtual machine, and the resulting offsets can be provided to the open source or self-developed virtual machine introspection tools to complete the process of semantic reconstruction. The VMOffset prototype system is implemented on the KVM (kernel-based virtual machine) virtualization platform, and the effectiveness and performance of VMOffset are experimentally analyzed based on virtual machines of different kernel

* 基金项目: 国家自然科学基金(U19A2081, 61802270); 国家“双创”示范基地之变革性技术国际研发转化平台资助项目(C700011); 四川省重点研发项目(2018G20100)

Foundation item: National Natural Science Foundation of China (U19A2081, 61802270); Transformational Technology Int’l Research platform for National Dual Innovation Base (C700011); Key Research Projects in Sichuan (2018G20100)

收稿时间: 2018-12-01; 修改时间: 2019-07-04; 采用时间: 2020-01-02

version operating systems. The results show that VMOffset can automatically complete the process-level semantic reconstruction process of each target virtual machine, and only introduces the performance loss within 0.05% in the startup phase of the target virtual machine.
Key words: virtual machine introspection; semantic reconstruction; offset; virtual machine monitor; portability

虚拟机自省(virtual machine introspection,简称 VMI)机制最早于 2003 年由 Tal Garfinkel 和 Mendel Rosenblum 在 Livewire[1]中提出,VMI 通过在目标虚拟机(target virtual machine,简称 TVM)外部获取 TVM 的底层状态,如 CPU 寄存器、内存、存储设备等,可有效地监控或干预 TVM 的运行.随着软硬件技术的高速发展,以 KVM、XEN、VMWare 等为代表的系统虚拟化技术极大地推动了由多任务计算到多操作系统计算的发展,已成为云计算架构中关键的抽象层次和重要的支撑技术[2].传统的安全软件置于 TVM 内部,因与恶意软件运行于同一特权级,易被其绕过或攻击而失效[3].VMI 基于虚拟化技术,利用虚拟机监视器(virtual machine monitor,简称 VMM)的高特权级与强隔离性,能够实现安全软件与恶意软件的分离,具有隔离性、自省性与干预性[4],在恶意软件分析[5]、入侵检测系统[1]、内核完整性检测[6]等安全方面得到广泛应用.VMI 在 TVM 外部获取的底层状态为二进制数据,其代表的操作系统级语义是无法直接知晓的,这种语义差异被称为 VMI 的语义鸿沟(sematic gap)[7]问题.将得到的二进制底层数据转换为可理解的高层语义的过程,即为语义重构.

语义重构是 VMI 技术的关键步骤.现有工作从不同层次、不同角度进行了研究与尝试,以期准确、高效地进行语义重构,得到TVM中的信息,实现对TVM的安全监控.根据语义重构的方法及依赖条件,可将现有解决方案分为以下几类.

(1) 基于目标虚拟机.

根据进行语义重构的位置,又可将此种方案分为两类.

1) 直接在 TVM 中获取语义信息.如 X-TIER[8]在 TVM 中添加内核模块,借助 TVM 本身获取丰富的语义信息,但其自身容易受到攻击;
2) 借助 TVM 内部信息在 TVM 外部进行语义重构.如:开源内存取证框架 Volatility[9]通过在 TVM 内部编译文件获取内核的相关信息,基于这些信息对 TVM 的内存镜像进行离线解析得到高层语义;Libvmi[10]基于 XenAccess[11]可实现对 TVM 内存及寄存器信息的实时读写,但其需要在 TVM 中加载内核模块得到进程结构体关键成员相对于结构体首地址的偏移量(下文中均简称为进程偏移量)来定位解析 TVM 内存中的关键信息;vDetector[3]基于 AntFarm[12],通过硬件架构知识在 VMM 层获取 TVM 中真实的进程视图,其同样需要在 TVM 中加载内核模块得到进程偏移量,以便在 TVM 外部实现进程语义的重构.

(2) 基于安全虚拟机(secure virtual machine,简称 SVM).

其中,文献[13]在 SVM 中同步模拟 TVM 中待监控进程的执行,完成对 TVM 进程信息的获取与监控.Virtuoso[14]将如 ps 等的待检测指令在 SVM 内执行多次,抽取与运算相关的指令生成指令路径,通过路径片段抽取、融合、转译,形成可在 TVM 外执行语义重构的代码.该方法需反复训练,性能损耗较大.VMST[15]对 Virtuoso 进行改进,利用内核重定向技术将数据访问重定向到 TVM 内存中生成自省程序.文献[16]结合了 Virtuoso[13]和 VMST[14],在性能方面有了较大的提升,但其需要两个与 TVM 内核版本相同的 SVM 进行自省操作,消耗了宿主机资源.

(3) 基于内核源码或内存转储文件.

如 KOP[17]基于 points-to[18]对内核的源代码进行分析构造拓展类型图,可在内存快照中映射内核数据结构.改进的 InSight[19]基于内核源码建立指针类型之间的 used-as 关系,实现内核数据结构中指针声明类型至其实际用途及目标地址的转换,以处理 TVM 内核的动态指针操作简化 VMI 程序的开发.VMwatcher[20]从源码中获取内核数据结构定义作为模板,得到进程偏移量等信息重构外部语义视图.Min-C[21]基于内核源码自动生成语义重建库,并结合其构造的 C 解释器定位解析 TVM 的相关数据结构获取高层语义.Volatilinux[22]则从内存转储文件中得到内核结构偏移,并基于此重构出 TVM 的进程级语义信息.

上述语义重构方案均存在对 TVM 内核版本敏感的问题,对于不同内核版本的 TVM,上述(1)类方法需要再

次访问 TVM 并手动配置进程偏移量信息;(2)类方法需要安装与 TVM 内核版本一致的 SVM 完成自省;(3)类方法则需要重新分析内核源码,导致系统的通用性及可移植性差,自动化程度低.此外,基于 TVM 的 VMI 方法可能因不具备权限或源码缺失等情况无法加载内核模块,导致自省无法实施.基于 SVM 的 VMI 方法需配置 SVM,对自省环境的要求高,且性能损耗较大.基于内核源码的 VMI 方法需分析及搜索的信息较多,自省效率不高,且分析内存转储文件无法获取运行状态 TVM 的信息.针对这些不足,本文基于 KVM-QEMU 虚拟化平台,提出了一种虚拟机自省中的语义重构改进方法 VMOffset(virtual machine offset),主要贡献如下:

(1) VMOffset 解决了上述 VMI 方法实现过程与 TVM 内核版本相关的问题.在不需要知道 TVM 内核版本的情况下,VMOffset 可基于结构体成员自身属性制定约束条件获取 TVM 的进程偏移量信息.该方法屏蔽了 TVM 内核版本的差异性,无需访问 TVM,对 TVM 无侵入性,且无需额外引入 SVM,具有通用性及可移植性;

(2) VMOffset 在 TVM 启动阶段完成进程偏移量的获取,性能损耗小,且启动阶段不会出现内核 Rootkit 等攻击,保障了所得信息的安全性.此外,VMOffset 的实现过程不依赖特定内核源码,TVM 内核源码的修改不会影响结果的正确性与可靠性;

(3) VMOffset 可将所得进程偏移量提供给自研或开源 VMI 工具完成语义重构过程,无需手动进行信息配置,自动化程度高,且可扩展 VMI 开源工具以实现虚拟机隐藏进程检测,进程代码段完整性度量功能,增强虚拟化环境下的安全能力.

本文第 1 节介绍 VMOffset 系统的设计及架构.第 2 节阐述关键技术.第 3 节对 VMOffset 进行测试评估.第 4 节总结全文.

1 VMOffset 系统设计

将在 TVM 外部提取的底层状态数据重构为高层语义信息的关键是获取两类信息:(1) 计算机组织、存储待重构数据元素的方式,如链表、树等;(2) 待重构数据元素的三元组信息⟨数据元素类型,起始地址,偏移量⟩[2].同样地,VMOffset 通过获取这两类信息实现语义重构过程,其设计基于以下前提:(1) Linux 操作系统通常使用链表组织关键信息,如进程、文件、内核模块等;(2) 虽然内核数据结构成员数目及布局会因内核版本的不同而不同,但部分关键属性成员是一直存在的,且其类型不变.以进程为例,如图 1 所示,Linux 内核使用 *task_struct* 结构体描述单个进程的详细信息,并使用双向循环链表将各个进程的 *task_struct* 组织起来.该结构体中部分关键成员是一直存在的,如成员 *pid* 为进程号,用于标识各个进程;成员 *comm* 可描述进程名称;成员 *tasks* 用于链接整个进程链表.可知:若得到某个进程 *task_struct* 结构体的起始地址,根据 *pid* 及 *comm* 的偏移量,就可重构出该进程的进程号及进程名;同理,根据 *tasks* 成员的偏移量,即可通过其链表指针得到下一个进程 *tasks* 成员的地址,依次遍历可得到 TVM 中的所有进程信息.

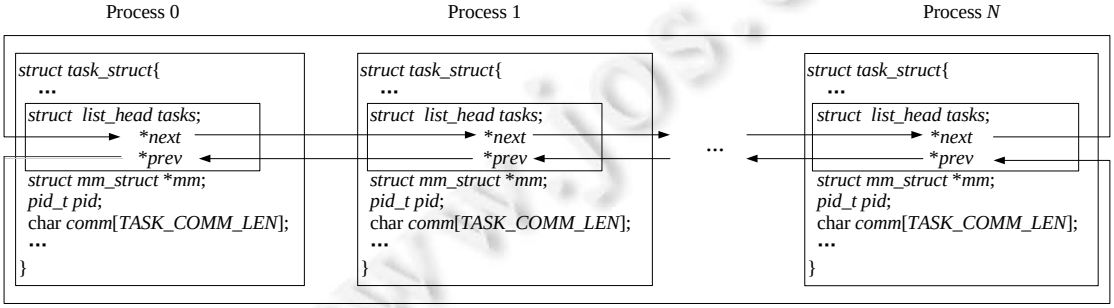


Fig.1 Structure of process linked list in Linux

图 1 Linux 进程链表结构

在该种语义重构方法中,不同于多数 VMI 工具需依赖 TVM 本身或内核源码等得到进程偏移量,VMOffset 可屏蔽 TVM 内核版本的差异性,实现在 TVM 外部对 TVM 中进程偏移量的自动获取,并基于此完成 TVM 进程级语义的重构.VMOffset 基于公式(1)及公式(2)设计实现:

$$MemAddr_{tsA}-TaskAddr_{tsA}=MemAddr_{tsB}-TaskAddr_{tsB},\forall A,B\in Prolist \tag{1}$$

$$Member\vDash ConditionSet,\forall Member\in Task\_Struct \tag{2}$$

如公式(1)所示,其中,*Prolist* 为进程链表,*A*、*B* 表示进程链表中的任意进程实例,$MemAddr_{tsA}$ 与 $MemAddr_{tsB}$ 分别表示 *A* 和 *B* 进程描述符中某相同成员的地址,$TaskAddr_{tsA}$、$TaskAddr_{tsB}$ 则分别为 *A*、*B* 进程描述符的起始地址.由公式(1)可知:对 TVM 进程链表中的任意不同进程实例而言,进程描述符 *task_struct* 结构体某一成员的地址相对于进程描述符首地址的偏移量是不变的,即,进程偏移量对各个进程实例而言是相同的.

如公式(2)所示,其中,*Task_Struct* 为进程描述符,*Member* 表示进程描述符中的某个成员,*ConditionSet* 为特定约束条件的集合.因进程描述符中各个成员均有其特定于自身的意义,因此会满足基于其自身属性的相关约束条件.如 *pid* 作为各个进程的进程号,具有标识进程的作用,因此各个进程的 *pid* 均唯一,即:*pid* 成员需满足对不同进程实例而言,其值必须不同的约束条件.

VMOffset 基于上述思想完成 TVM 进程偏移量的自动获取,其总体架构如图 2 所示.

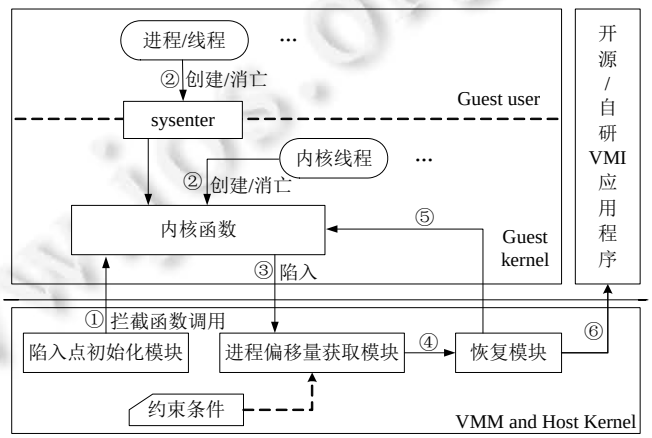


Fig.2 Overall architecture of VMOffset

图 2 VMOffset 总体架构

图 2 中 VMOffset 各个模块的功能与总体流程如下.

(1) 陷入点初始化模块:该模块在 VMM 中透明修改 TVM 中特定内核函数的指令代码,拦截 TVM 中特定内核函数调用,以监控 TVM 中进程/线程的创建及消亡事件,使其能够产生 VM-EXIT 陷入至 VMM 中,以此触发后续获取进程偏移量的相关流程;

(2) 进程偏移量获取模块:该模块基于陷入点,在 VMM 中获取指定监控事件发生后 TVM 的寄存器状态及内存状态等信息,依照公式(2),结合基于成员自身属性制定的约束条件获取进程偏移量.由公式(1)可知:对不同进程实例而言,其相同成员的偏移量是不变的.因此,TVM 中不同进程实例因产生特定监控事件触发 VM-EXIT 陷入 VMM 时,均可执行相同的流程以获取进程偏移量.该模块获取偏移量的进程结构体成员包括:可表示进程基本信息的 *pid*、*comm*,用于遍历进程链表的 *tasks*,用于管理进程内存信息的 *mm*,用于查找进程页表访问进程指定地址内容的 *mm.pgd* 以及可用于获取进程代码段内容的 *mm.start_code* 及 *mm.end_code*;

(3) 恢复模块:因获取偏移量的各个成员属性不同,各自依赖的约束条件也不尽相同,因此,并非某一次操作即可获取全部所需进程偏移量.故对于每次 VM-EXIT,完成进程偏移量获取模块的相关操作后,恢

复模块判断是否已获取全部所需进程偏移量:若尚未获取完毕,则进行相关模拟操作后执行 VM-ENTRY 进入 TVM 完成内核函数的正常执行流程;否则恢复对 TVM 中相关内核函数的修改,使进程/线程的创建及消亡动作不再产生 VM-EXIT 陷入至 VMM,减少了对 TVM 的影响,并将所得进程偏移量提供给开源或自研的 VMI 应用程序,完成 TVM 的自省过程.

2 VMOffset 关键技术

2.1 拦截内核函数调用的事件监控机制

VMOffset 通过监控 TVM 中进程创建及消亡的内核函数调用事件,触发 VMM 中进程偏移量的获取流程,并基于此触发点得到 TVM 的状态信息,作为后续获取进程偏移量的条件与依据.Linux 用户进程的创建通过系统调用 *fork*(·)、*clone*(·)和 *vfork*(·)实现,进程的消亡通过系统调用 *exit*(·)、*wait*(·)实现.

可通过修改 *SYSENTER_CS_MSR* 寄存器[23]或系统调用表[24]等方式拦截上述系统调用,以监控进程的创建及消亡.但该种方式存在如下缺陷.

1) 系统调用是用户进程访问内核服务的一个接口,内核线程直接通过调用内核函数使用内核提供的服务.因此在 TVM 系统启动阶段,产生用户进程之前,并不会发生系统调用事件,故该种方式无法监控 TVM 整个时期的进程创建及消亡动作;
2) 在虚拟化环境下,为了在 VMM 层捕获到 TVM 中的系统调用事件,通常需要设置虚拟机控制结构体的相关字段[24],用于在 VMM 中捕获缺页异常或一般保护异常.这样会引入大量与系统调用无关的异常陷入,对 TVM 系统性能带来较大影响.

针对上述不足,VMOffset 通过监控特定内核函数的调用来感知 TVM 中进程创建及消亡的动作,这种方式可监控 TVM 各个阶段的进程创建及消亡动作,且不会引入无关的异常陷入.系统调用 *fork*(·)、*clone*(·)和 *vfork*(·)的服务例程分别为 *sys_fork*(·)、*sys_clone*(·)和 *sys_vfork*(·),这 3 个函数最终都会调用 *do_fork*(·)函数实现具体创建工作.系统调用 *exit*(·)和 *wait*(·)的服务例程分别为 *sys_exit*(·)、*sys_wait*(·),分别调用内核函数 *do_exit*(·)及 *do_wait*(·),这两个内核函数最终都会调用函数 *release_task*(·)释放待消亡进程的进程描述符.因此,VMOffset 通过监控内核函数 *do_fork*(·)及 *release_task*(·)的调用,实时感知 TVM 中进程创建及消亡事件,其实现原理如图 3 所示.

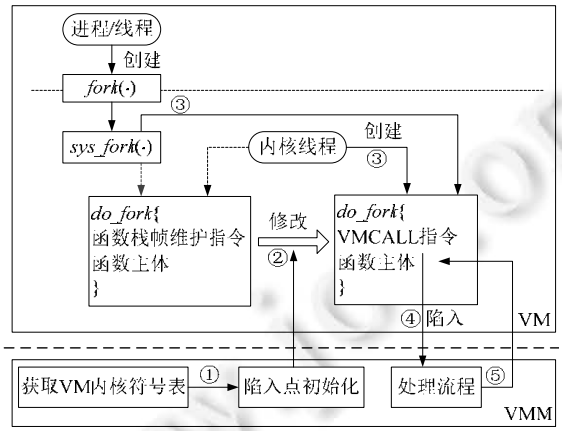


Fig.3 Principle of event monitoring

图 3 事件监控原理

函数之间的调用通过栈来实现,函数执行时所用栈被称为函数栈帧,EBP 寄存器作为帧指针,指向函数栈帧的开始位置;ESP 寄存器作为栈指针,则指向栈顶位置.当调用函数以 call 方式调用被调用函数时,call 指令首先将调用函数 call 指令的下一条指令地址压入栈中,然后将被调用函数地址保存至 EIP 寄存器.执行流程从 EIP

寄存器指定地址开始到达被调用函数,被调用函数首先通过“pushl%ebp;movl%esp,%ebp”保存调用函数的栈帧信息,便于被调用函数执行完毕后还原调用函数栈帧信息.因此,为维护函数栈帧,以 call 方式调用的函数的开头位置均为三字节的“pushl%ebp;movl%esp,%ebp”指令.由 Intel 手册可知:可陷入敏感指令 VMCALL 机器码为“.byte 0x0f,0x01,0xc1”,长度也为三字节.基于此,VMOffset 通过指令替换来感知 TVM 的内核函数调用行为.为了让不同的内核功能单元更好地协同工作,Linux 内核编译过程中会生成 System.map 文件存放内核符号表,其中包含所有的全局内核项(函数及变量)的地址,可在表中查找符号名得到对应的符号地址.VMOffset 参考开源项目 OpenCIT[25],在 TVM 外部挂载 TVM 镜像得到其内核符号表,随后从内核符号表中读取内核函数 *do_fork*(·)及 *release_task*(·)的入口地址,传至陷入点初始化模块.如图 3 所示,该模块在 TVM 启动之前,在 VMM 层将上述内核函数起始三字节的函数栈帧维护指令替换为 VMCALL 指令,该操作对 TVM 透明,无需在 TVM 中添加代理模块.此后,TVM 中发生进程/线程创建及消亡的事件时,就会因 VMCALL 指令陷入至 VMM 中,VMM 中完成获取进程偏移量的处理流程之后,模拟 TVM 中函数栈帧维护行为,之后返回 TVM 继续执行.待所需进程偏移量全部获取完毕后,VMOffset 恢复对内核函数的修改,使其不再陷入,减少了对目标虚拟机的影响.

2.2 基于迭代的进程偏移量获取方法

VMOffset 拦截 TVM 中内核函数 *do_fork*(·)及 *release_task*(·)的调用事件后,以 TVM 的寄存器状态、内存状态信息及相关约束条件为依据,基于迭代的方法进行进程偏移量的获取.首先定义相关实体如下.

- XF_{Candi_init}={*xf*|*xf* 为初始状态下成员 *X* 的偏移量候选项};
- XF_{Candi_prev}={*xf*|*xf* 为执行偏移量获取流程前成员 *X* 的偏移量候选项};
- XF_{Candi_not}={*xf*|*xf* 为陷入时刻内存状态不满足成员 *X* 约束条件的偏移量候选项};
- XF_{Candi_after}={*xf*|*xf* 为执行偏移量获取流程后成员 *X* 的偏移量候选项}.

根据上述定义,可知执行一次偏移量获取流程后,有:$XF_{Candi\_after}=XF_{Candi\_prev}-XF_{Candi\_not}$.

基于上述定义及 VMOffset 设计思想,图 4 以伪代码的方式给出了 VMOffset 获取进程结构体某特定成员偏移量的算法过程,其中部分变量及函数的说明见表 1.

Algorithm. *get_member_offset.*

Input: *vcpu.esp*, *specific_size*;
Output: *member offset*.

Phase-1: Get initial *candi_offset_list*

1. **for** *candi_offset*=0; *candi_offset*≤*len*(*struct task_struct*) **do**
2. *candi_offset_list*←*list_add*(*candi_offset_list*,*candi_offset*);
3. *candi_offset*←*candi_offset*+*specific_size*;
4. **end for**

Phase-2: Filtrate to get the correct *member_offset*

5. **do**
6. *ts_addr*←*get_ts_addr*(*vcpu.esp*);
7. **for each** *candi_offset* in *candi_offset_list* **do**
8. *offset_content*←*read_addr*(*ts_addr*+*candi_offset*);
9. **if** *check_condition*(*offset_content*)=false **then**
10. *candi_offset_list*←*list_del*(*candi_offset_list*,*candi_offset*);
11. **end for**
12. **while** *count_num*(*candi_offset_list*)=1;
13. *member_offset*←*get_member_from_list*(*candi_offset_list*);

Fig.4 Process of acquiring process offset

图 4 进程偏移量获取过程

Table 1 Related description of process offset acquisition algorithm

表 1 进程偏移量获取算法相关说明

变量/函数名	说明
candi_offset_list	该变量为链表,链表中包含该特定成员的所有偏移量候选项
get_ts_addr(·)	该函数用于获取 TVM 中当前进程描述符的起始地址,具体见第 2.2.1 节
read_addr(·)	该函数用于读取 TVM 中某地址处的具体内容,具体见第 2.2.2 节
check_condition(·)	该函数用于判断相关内存内容是否满足制定的约束条件,具体见第 2.2.3 节

图 4 中:

- 第 1 行~第 4 行首先依据待获取偏移量的成员类型得到其偏移量的所有候选项,即 XF_{Candi_init}.以 *pid* 为例,Linux 使用 4 字节的 int 类型存储进程的 *pid*,根据结构体的字节对齐原则可知,成员 *pid* 相对于 *task_struct* 结构体起始地址的偏移量必须是 4 的倍数,故设置其 *specific_size* 为 4.需注意的是:TVM 内核源码可能会被修改,因此从 0 开始到 *task_struct* 结构体大小的范围内,每增加 *specific_size* 的值均可能为 *pid* 成员的偏移量,将这些值均作为成员 *pid* 的偏移量候选项添加至集合中;
- 第 5 行~第 13 行则对所有候选项进行筛选,得到最终偏移量.其中,第 6 行~第 11 行为一次偏移量获取的流程.
  - 第 6 行根据陷入时刻 TVM 的寄存器信息获取当前进程结构体的首地址;
  - 第 7 行~第 11 行则对集合中的每个偏移量候选项,首先获取该地址处的内存内容,然后以基于该成员属性制定的约束条件为依据,判断其内容是否满足约束条件:若不满足,则将该候选项从集合中删除.

  第 1 次执行偏移量获取流程时,XF_{Candi_prev} 即为 XF_{Candi_init},依照条件删除 XF_{Candi_not} 后,可得 XF_{Candi_after}.由于对不同进程实例而言,其相同成员的偏移量是不变的,因此,当 TVM 中不同进程实例因产生指定监控事件陷入到 VMM 中时,VMOffset 均可以上一次得到的 XF_{Candi_after} 为此次获取流程的 XF_{Candi_prev},执行第 6 行~第 11 行的偏移量获取流程,删除此次内存状态不满足约束条件的 XF_{Candi_not} 集合,得到新的 XF_{Candi_after}.迭代地执行此过程,直到对应集合中仅有一个成员,此时说明已成功获取该结构体成员的偏移量,将其输出.

在 TVM 启动过程中,对某特定成员而言,TVM 中各个进程实例在该成员的各个候选偏移量处的内存内容是动态变化的,其内存内容包括满足和不满足所制定约束条件两种可能.将候选偏移量处内存内容满足约束条件的事件表示为 *T*,以 P_T 表示事件 *T* 发生的概率.可知:对某成员唯一正确的偏移量而言,其内存内容会始终满足基于其自身制定的约束条件,故其 P_T 始终为 1;而对该成员的其他候选偏移量而言,其内存内容则会逐渐表现出该处成员自身特性,而非约束条件所属成员特性,因此,其 P_T 小于 1.若事件 *T* 发生的概率 P_T 等于 1,则 *n* 次迭代过程中事件 *T* 一直发生的概率 $(P_T)^n$ 也始终为 1,即正确的偏移量在迭代过程中会始终保留不被删除.由公式(3)可知:当事件 *T* 发生的概率 P_T 小于 1 时,*n* 次迭代过程中事件 *T* 一直发生的概率 $(P_T)^n$ 会逐渐减小至接近 0:

$$\lim_{n\to\infty}(P_T)^n = 0, 0 < P_T < 1 \tag{3}$$

一般而言,TVM 启动过程中调用 *do_fork*(·)及 *release_task*(·)的次数超过 1 000 次.因此,随着上述过程迭代次数的增加,不符合约束条件的候选偏移量会被逐渐删除,最终得到唯一正确的偏移量.

2.2.1 获取进程描述符起始地址

如图 5 所示,Linux 内核中,默认情况下内核栈占据两个连续的物理页面,即 8KB.内核栈由高地址向低地址增长,8KB 地址区间的起始位置保存着线程描述符 *thread_info*.当 TVM 中进程因调用 *do_fork*(·)及 *release_task*(·)陷入至 VMM 时,TVM ESP 寄存器指向内核栈的栈顶单元,通过屏蔽栈顶地址的低 13 位,可获取线程描述符 *thread_info* 结构体的首地址,其第 1 个成员 *task* 指向当前进程的进程描述符 *task_struct*,故可基于此获取进程描述符 *task_struct* 结构体的起始地址.

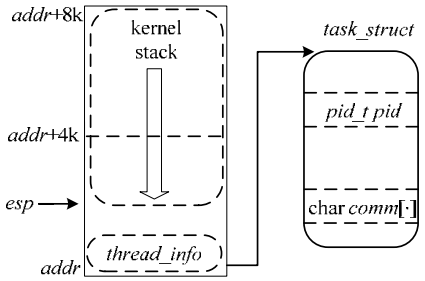


Fig.5 Relationship between kernel stack and process descriptor

图 5 内核栈与进程描述符的关系

2.2.2 获取目标虚拟机内存内容

为了让虚拟机使用一段隔离的、从 0 开始且连续的内存空间,内存虚拟化机制引入了一层新的地址空间,即虚拟机物理地址空间.如图 6 所示,虚拟机内存地址访问时存在两次地址转换:虚拟机页表完成虚拟机虚拟地址 GVA(guest virtual address)到虚拟机物理地址 GPA(guest physical address)的转换,该转换过程中得到的虚拟机各级页目录项地址及最终的 GPA 均不能直接发送到真实物理机的系统总线,需通过 VMM 的扩展页表 EPT (extended page table)将其转换为宿主机物理地址 HPA(host physical address).因此,VMOffset 通过在 VMM 中实现上述地址转换流程,完成对 TVM 中指定 GVA 的内存访问,获取其对应的内容.

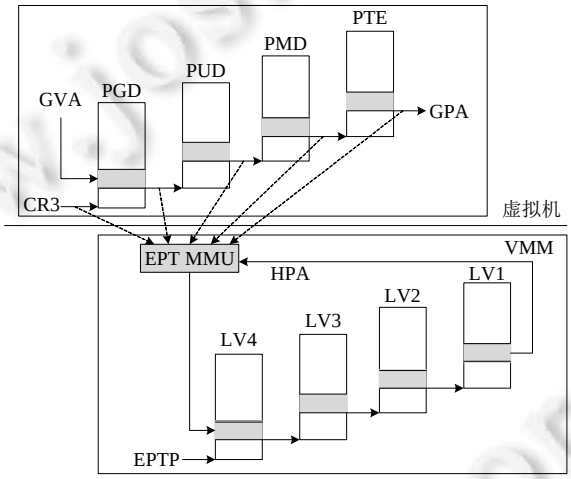


Fig.6 Virtual machine address translation process

图 6 虚拟机地址转换流程

2.2.3 基于成员属性的约束条件制定

VMOffset 基于结构体成员自身属性制定相关约束条件,并依据约束条件对各个成员的偏移量候选项进行筛选.具体地,如公式(4)所示:定义 XF_{candi} 为成员 X 所有偏移量候选项的集合,xf 为集合中的每个候选偏移量,$Addr_{ts}$ 为 TVM 发生 VM-EXIT 时当前进程 *task_struct* 结构体的起始地址,$Con(Addrts+xf)$为其 xf 偏移处的内存内容,*ConditionSet* 为成员 X 的约束条件.VMOffset 基于第 2.2 节所述迭代方法,依据 *ConditionSet* 逐步缩小 XF_{candi} 集合的成员数目得到成员 X 的最终 xf:

$$XF_{candi}=\{xf|Con(Addr_{ts}+xf)\in ConditionSet\} \tag{4}$$

基于公式(4),针对不同的成员,具体化约束条件与筛选方法如下.

1) 成员 *tasks*.

tasks 成员为进程链表节点,在未遭受内核 Rootkit 攻击的情况下,*tasks* 所在链表的成员数目应与虚拟机中

真实进程总数一致.基于此,制定其约束条件如下:

$$TF_{candi}=\left\{tf\left|\begin{array}{c}Num_{real}=Num_{tf},\\ Num_{real}=Count_{fork}-Count_{exit},Num_{tf}=Traverselinklist(Con(Addr_{ts}+tf))\end{array}\right.\right\} \tag{5}$$

公式(5)中,Num_{real}为 TVM 中真实进程数目,通过 VMM 拦截的 *do_fork*(·)的调用次数 $Count_{fork}$及 *release_task*(·)的调用次数 $Count_{exit}$得到.将 $Con(Addr_{ts}+tf)$视为一个进程链表节点,依次遍历即可得到该节点所在链表的成员总数 Num_{tf}.基于公式(5)筛选 *tf* 的过程中需注意:

(1) Linux 内核的第 1 个进程 swapper 由内核自身由无到有创建,其使用静态分配的数据结构,不通过调用内核函数 *do_fork*(·)得到.该进程作为 Linux 内核的祖先进程,负责创建 Linux 中的其他进程完成内核的初始化;

(2) 当 VMM 中拦截到 *do_fork*(·)及 *release_task*(·)的调用事件时,函数的具体内容尚未执行,因此当前时刻,TVM 中进程的数目并未改变;待函数执行完毕后,TVM 中进程总数才会增加或者减少.如 VMM 中第 1 次拦截到 *do_fork*(·)时,可知 TVM 中当前进程为 swapper 进程,且该时刻 TVM 中仅有 1 个进程,即 Num_{real}为 1.

2) 成员 *pid*.

pid 成员作为进程的标识,其约束条件可用公式(6)表示:

$$PF_{candi}=\left\{pf\left|\begin{array}{c}Con_i(Addr_{ts}^i+pf)\neq Con_j(Addr_{ts}^j+pf),\\ 0\leqslant Con_i(Addr_{ts}^i+pf)<PidLimit,PidLimit<32768,\forall i,j\in Prolist,i\neq j\end{array}\right.\right\} \tag{6}$$

其中,*Prolist* 为 TVM 中进程链表,*i*、*j* 分别为链表中某个进程.公式(6)表示对进程链表中的任意不同进程,其 *pid* 的值都是不同的,且 *pid* 均处于 0~*PidLimit* 的范围内.Linux 系统一般从 0 开始,依次连续分配 *pid* 直到可分配的 *pid* 上限,之后则重新从某个值开始依次连续查找未被使用的 *pid* 进行分配.默认情况下,不同系统的 *pid* 上限是不同的,且可被修改.但是由于内核初始化阶段分配的 *pid* 均较小,且部分系统的 *pid* 上限默认为 32 768,因此 VMOffset 将 *PidLimit* 设为 32 768.基于此,对 *pf* 进行筛选.

3) 成员 *comm*.

comm 成员用于描述进程名称.如前所述,Linux 内核静态初始化其第 1 个进程的 *task_struct* 结构体内容,其中,进程名通过宏定义为“swapper”,可通过匹配“swapper”字符串筛选 *comm* 的偏移量候选项.但若内核源码被修改,“swapper”被改为其他任意字符串,则此种匹配方式得到的偏移量将是错误的,进而会导致自省结果发生错误.因此,VMOffset 基于其自身属性制定约束条件.如公式(7)所示:成员 *comm* 表示进程名称,故其每个字符的 ASCII 码均在可显示字符的 ASCII 码范围内,且该字符数组的内容不可为空,并以字符‘\0’结尾:

$$CF_{candi}=\left\{cf\left|\begin{array}{c}Con_i(Addr_{ts}^i+cf)[j]\in ShowCharSet,\\ Con_i(Addr_{ts}^i+cf)[end]=\text{'\textbackslash 0'},\forall i\in Prolist,\forall j\in[0,commlen),commlen>0\end{array}\right.\right\} \tag{7}$$

公式(7)所描述的约束条件不会因内核源码的修改而发生变化,VMOffset 基于此筛选 *cf*.

4) 成员 *mm*.

mm 成员用于描述进程虚拟地址空间信息.由内核源码可知:一般而言,*mm* 成员的下一个成员即为 *active_mm*.而 swapper 进程的 *active_mm* 成员由内核静态初始化为 *init_mm* 的地址,该地址可由 TVM 内核符号表得到,可基于此获取 *mm* 的偏移量.但是为了保证内核源码被修改后所得偏移量的正确性,VMOffset 不以成员位置关系为依据,而是基于成员自身属性进行约束条件的制定.成员 *mm* 的特点如公式(8)所示:Linux 中,内核线程的 *mm* 成员为 0,用户进程 *mm* 成员的值则与 *active_mm* 成员的内容一致:

$$MF_{candi}=\{mf\mid Con_i(Addr_{ts}^i+mf)=0\,\|\,Con_i(Addr_{ts}^i+mf)=Con_i(Addr_{ts}^i+amf),\forall i\in Prolist\} \tag{8}$$

公式(8)中,*amf* 为成员 *active_mm* 的偏移量,*Prolist* 的含义同公式(6).

5) 成员 *pgd*.

mm_struct 结构体中成员 *pgd* 用于存储进程的页目录基地址,该值对应的物理地址会被载入 CR3 寄存器,

用于遍历进程页表实现地址转换功能.*pgd* 的值与 CR3 寄存器的值相差 *page_offset*,32 位下,*page_offset* 为 0x*c*0000000;64 位下,则为 0x*ffffffff*80000000.因此,其约束条件可由公式(9)描述:

$$PGF_{candi}=\{pgf\mid Con_i(Addr_{ms}^i+pgf)=Vcpu.cr3+page\_offset,\forall i\in UserPro\} \tag{9}$$

其中,*i* 为某个用户进程; $Addr_{ms}^i$ 为该用户进程 *mm_struct* 结构体的起始地址,可由 $Addr_{ts}^i$ 与 *mm offset* 得到;*Vcpu*.*cr*3 为 TVM 陷入 VMM 时,其 CR3 寄存器的值.

6) 成员 *start_code* 及 *end_code*.

进程代码段在内存中的起始地址与结束地址分别存储在 *mm_struct* 结构体的 *start_code* 及 *end_code* 域中.IA-32 体系下,用户进程虚拟地址空间布局如图 7 所示.

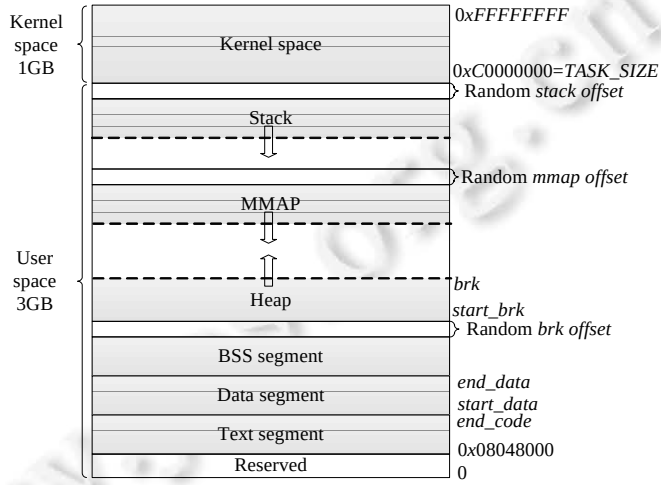


Fig.7 Process virtual address space layout of IA-32 architecture

图 7 IA-32 进程虚拟地址空间布局

进程代码段为图 7 中所示的 text segment,每个体系结构为 text 段制定了一个特定的起始地址:*x*86 架构下始于 0*x*08048000,*x*86_64 下则为 0*x*400000.基于此,可得到成员 *start_code* 的偏移量.一般而言,内核源码中成员 *start_code* 与 *end_code* 处于相邻位置.同样地,为保证所得偏移量的可靠性,VMOffset 以 *end_code* 成员属性为依据获取对应偏移量,其约束条件可由公式(10)描述:

$$ECF_{candi}=\left\{ecf\left|\begin{array}{c}\mathrm{Min}Limit\leqslant Con_i(Addr_{ms}^i+ecf)<\mathrm{Max}Limit,PageAccess(Con_i(Addr_{ms}^i+ecf))=RO\,\&\,X\\ \mathrm{Min}Limit\geqslant Con_i(Addr_{ms}^i+scf),\mathrm{Max}Limit<TASK\_SIZE,\forall i\in UserPro\end{array}\right.\right\} \tag{10}$$

其中,*i*、*UserPro* 及 $Addr_{ms}^i$ 的含义同公式(9);*scf* 为 *start_code offset*;*TASK_SIZE* 为一常数,用于将进程虚拟地址空间划分为内核空间和用户空间.由图 7 可知,进程代码段位于用户空间.

公式(10)中, $PageAccess(Con_i(Addr_{ms}^i+ecf))$ 为 *ecf* 偏移处内容所在页面的权限,进程代码段所在页面的权限为只读与可执行.基于此,可进一步更新公式(10)中的 Min*Limit* 与 Max*Limit*,对 *ecf* 进行筛选.此处以 IA-32 体系结构为例,给出筛选 1 次 *ecf* 的伪代码,如图 8 所示.

图 8 中:第 1 行~第 9 行基于进程虚拟地址空间布局及代码段所在页面权限更新 Min*Limit* 和 Max*Limit*,选取具有可读写或不可执行权限的最小 *offset_content* 作为 Max*Limit*,同时选取具有只读且可执行权限的最大 *offset_content* 作为 Min*Limit*;第 10 行~第 14 行则基于已更新的 Min*Limit* 与 Max*Limit*,删除 *offset_content* 不在此范围内的 *ecf*.图 8 为筛选 1 次 *ecf* 的过程,多次迭代可得到最终 *ecf*.

由上述基于成员属性制定约束条件,依照约束条件筛选各个成员偏移量候选项的过程可知,不同成员自身的约束条件与其所依赖条件均存在差异.如筛选 *tasks* 成员偏移量候选项的过程仅依赖 TVM 中真实进程数目;

而获取 *pid* 成员的偏移量则还依赖 *tasks offset* 遍历进程链表,得到所有进程 *pf* 偏移处的内容进行对比.因此,VMOffset 需基于成员自身属性与各个成员之间的相互联系,依次获取所需成员对应的偏移量.

Algorithm. *filtrate_end_code_offset*

Input: *ms_addr*, *start_code_offset*, *candi_ecf_list*, *TASK_SIZE*;
Output: new *candi_ecf_list*.

Phase-1: Traverse to narrow the limit

```
1. start_code←read_addr(ms_addr+start_code_offset);
2. min_limit←start_code, max_limit←TASK_SIZE;
3. for each candi_ecf in candi_ecf_list do
4.     offset_content←read_addr(ms_addr+candi_ecf);
5.     if offset_content∈(start_code,TASK_SIZE) then
6.         content_pte←get_pte(offset_content);
7.         if content_pte≠0 then
8.             update_limit(max_limit,min_limit,offset_content,content_pte);
9. end for
```

Phase-2: Delete *candi_ecf* beyond the limit to get the new *candi_ecf_list*

```
10.for each candi_ecf in candi_ecf_list do
11.    offset_content←read_addr(ms_addr+candi_ecf);
12.    if offset_content∉[min_limit,max_limit] then
13.        list_del(candi_ecf_list,candi_ecf);
14.end for
```

Fig.8 Algorithm of filtrating *end_code offset*

图 8 *end_code offset* 筛选算法

3 实验及结果分析

本节从功能及性能两个方面对 VMOffset 进行评测.功能测试用于验证 VMOffset 的有效性,即:能否屏蔽 TVM 内核版本的差异性,自动获取其进程偏移量,并基于此实现语义重构.性能测试用于验证 VMOffset 对 TVM 的性能影响.VMOffset 基于 KVM-QEMU 虚拟化平台实现,实验环境如下:宿主机为 Intel(R) Core(TM) i3-4160 双核 CPU,主频为 3.60GHz,物理内存为 4GB,支持硬件辅助虚拟化.操作系统为 64 位 CentOS7,内核版本 3.10.1,KVM 版本 kvm-kmod-3.10.1,QEMU 版本 Qemu-2.3.0.

3.1 功能测试

为验证 VMOffset 的内核版本无关性,部署了不同内核版本操作系统的 TVM,见表 2.

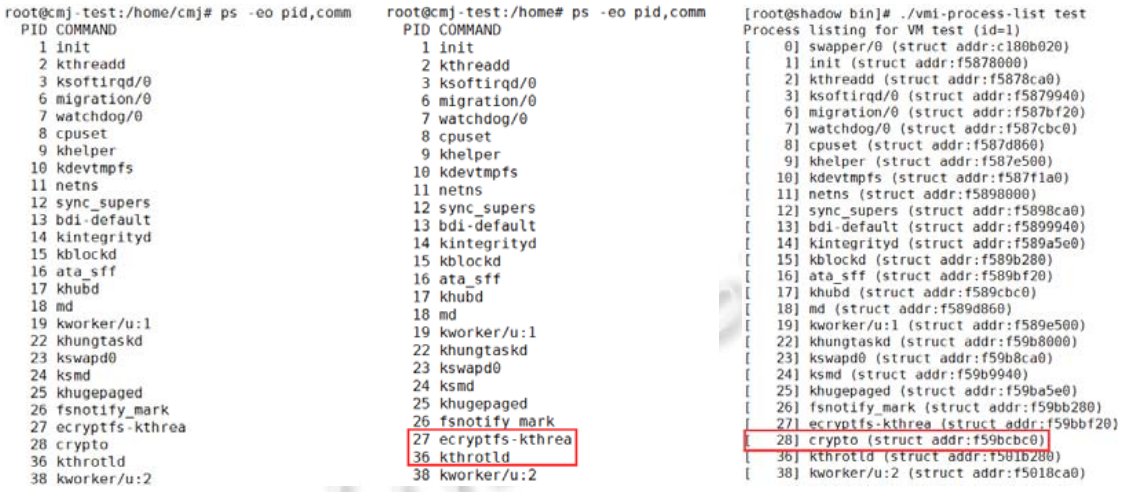
Table 2 Environment configuration of virtual machine

表 2 虚拟机环境配置

虚拟机	位数	操作系统	内核版本
VM1	64	CentOS6.5	2.6.32
VM2	32	Ubuntu12.04	3.2.0-24
VM3	64	CentOS7	3.10.0
VM4	64	Ubuntu14.04	4.4.0-31

实验中,TVM 的内核版本不同,对应的进程偏移量也不同.以成员 *pid* 的偏移量为例,CentOS6.5 中为 1 192,Ubuntu12.04 中为 516,CentOS7 中为 1 188,Ubuntu14.04 中为 1 064.在上述 TVM 内核版本未知的前提下,VMOffset 均能自动获取其进程偏移量,所得偏移量可提供给自研或开源 VMI 应用程序完成语义重构.为说明 VMOffset 对开源 VMI 工具的支持与兼容,本节以 VM2 为例,将 VMOffset 所得进程偏移量提供给 Libvmi,完成虚拟机自省过程.Libvmi 是一款具有代表性的开源 VMI 工具,专注于读写虚拟机内存,但其需要在 TVM 中加载内核模块获取进程偏移量,并将其手动写入配置文件中.本节将 VMOffset 所得进程偏移量提供给 Libvmi,使其无需进行手动获取及配置的操作.图 9 所示为 VMOffset 结合 Libvmi 对 TVM 中进程视图的自省结果,限于篇幅,

此处仅截取部分进程信息.图 9(a)为某一时刻 TVM 中进程视图,图 9(c)为同一时刻 VMOffset 在 TVM 外部重构所得进程视图,可见两者的结果是一致的.进程视图的重构可用于检测隐藏进程,因此本节在 TVM 中编译安装 Adore-ng 工具,隐藏 *pid* 为 28 的进程.隐藏后,TVM 中进程视图如图 9(b)所示,其中,28 号进程已被隐藏.而 VMOffset 重构视图仍与图 9(c)一致,其中显示了被隐藏的进程 crypto.可见:VMOffset 可自动实现 TVM 中进程信息的语义重构,且通过对比重构所得进程视图与 TVM 内部进程视图,可以有效地发现 TVM 中的隐藏进程.



(a) 初始 TVM 内部视图 (b) 隐藏进程后 TVM 内部视图 (c) 初始&隐藏进程后 TVM 外部重构视图

Fig.9 Process view introspection results of VMOffset

图 9 VMOffset 进程视图自省结果

除了重构 TVM 中的进程视图外,VMOffset 还可重构 TVM 中指定进程的代码段,代码段的获取可用于入侵检测,如通过验证代码段的完整性可判断该进程是否被恶意篡改.本节参考 VMPMS[26]提出的进程代码段分页式度量方法,基于 VMOffset 所得进程偏移量,在 TVM 外部开发用户层工具实现对 TVM 中指定进程的代码段完整性度量.以 Apache 服务进程 httpd 为例,图 10 为其度量结果.图 10(a)为 httpd 进程正常运行状态下的度量结果,基值通过对 SVM 中 httpd 进程代码段各页进行 SHA256 运算得到.图 10(b)为模拟攻击后 httpd 进程度量结果,对比可见:模拟攻击后进程名称和代码页地址等语义信息并无变化,但代码页的度量值发生了改变,如 126~128 等页.可见:VMOffset 可有效重构 TVM 中指定进程的代码段,且可实现其代码段的完整性度量,判断该进程是否被恶意篡改.

(a) 正常进程度量结果 (b) 模拟攻击后进程度量结果

Fig.10 VMOffset measurement result of process code segment in TVM

图 10 VMOffset 对 TVM 中进程代码段的度量结果

3.2 性能测试

本节通过相关测试评估 VMOffset 对 TVM 的性能影响,并考察 VMOffset 重构 TVM 进程视图及对 TVM 中进程代码段进行完整性度量的效率.最后对比了 VMOffset 与现有其他虚拟机自省工具,评估了 VMOffset 的有效性与通用性.

VMOffset 需在 TVM 启动阶段拦截其特定内核函数调用,以触发进程偏移量的获取流程.这一操作会导致本不会发生的 VM-EXIT,对 TVM 的启动带来影响.由于 VMOffset 在获取所需进程偏移量后会还原修改,因此本节通过测试 VMOffset 对 TVM 启动时间的影响,及 VMOffset 导致 TVM 发生 VM-EXIT 的次数与 TVM 启动阶段调用 *do_fork*(·)及 *release_task*(·)总次数的占比,来评估 VMOffset 对 TVM 启动阶段的性能影响.

表 3 为对 VM2 及 VM4 分别使用 VMOffset 获取 10 次进程偏移量所得测试结果的平均值,VM2 及 VM4 的配置信息见表 2.表 3 中,TVM 的启动时间通过开源工具软件 BootChart 测试得到,总次数为 TVM 启动阶段相关内核函数调用的总次数,以出现用户登录界面表示 TVM 启动阶段的结束.可见:对于 VM2,VMOffset 运行阶段导致的 VM-EXIT 次数仅占总次数的 3.74%,引入的时间损耗为 0.042%;对 VM4 的 VM-EXIT 占比则为 4.62%,引入的时间损耗仅为 0.041%.测试结果表明:VMOffset 在 TVM 启动完成之前即可获取全部所需进程偏移量,对 TVM 启动阶段引入的时间损耗在 0.05%以内.实验同时测试了 VMOffset 运行过程中,TVM 中出现的最大 *pid* 值,其中 VM2 出现的最大 *pid* 值为 63,VM4 中则为 120,均小于公式(6)中设定的 32 768,因此设定该上限值是可行的.此外,测试了 VMOffset 运行过程中部分成员偏移量候选项满足其约束条件的事件发生的次数,如 *task_struct* 中,*tasks*、*children*、*sibling* 均为链表节点.在 90 次 VM-EXIT 中,*children* 及 *sibling* 满足基于 *tasks* 制定的约束条件的次数仅为 2 次,其内存内容满足约束条件的事件发生的概率小于 1,因此多次迭代后均会被删除,从而得到 *tasks* 成员的真正偏移量.类似可知,多次迭代之后,VMOffset 可得到全部所需进程偏移量.

Table 3 Performance test result of TVM in start-up stage

表 3 TVM 启动阶段性能测试结果

虚拟机	启动时间/s (未部署 VMOffset)	启动时间/s (部署 VMOffset)	VM-EXIT 次数	总次数	VM-EXIT 占比/%	时间损耗/%
VM2	47.44	47.46	90	2 402	3.74	0.042
VM4	48.80	48.82	152	3 285	4.62	0.041

本节同时采用 UnixBench 微基准测试工具评估 VMOffset 对 TVM 运行时的性能影响,图 11 为部署与未部署 VMOffset 环境下,VM2 的系统性能测试结果.UnixBench 各测试项中,评分越高,表明性能越好.

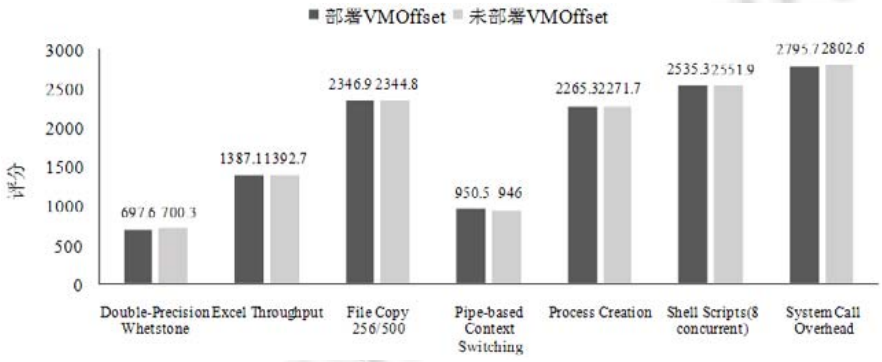


Fig.11 Performance test result of TVM in running stage

图 11 TVM 运行阶段性能测试结果

由图 11 可知,VM2 的进程创建与切换(process creation, pipe-based context switching)、系统调用(system call overhead, excel throughput)、shell 脚本测试(shell scripts)、文件传输(file copy)等方面的性能在部署与未部署 VMOffset 环境下均无明显波动.因 VMOffset 对 TVM 的性能影响主要源于拦截其内核函数调用导致的 VM-

EXIT损耗,且在获取所需进程偏移量后会还原修改使其不再陷入,故此后TVM的运行与未部署VMOffset无异.由实验结果可知:VMOffset仅对TVM的启动阶段引入0.05%之内的时间损耗,对其运行阶段无明显影响.

除了评估VMOffset对TVM启动阶段及运行阶段的性能影响,本节还测试了VMOffset重构TVM进程视图及对TVM中进程代码段进行完整性度量的效率.其中,图12展示了不同VMI方法重构TVM中进程视图所用时间的结果,表4则为VMOffset对TVM中不同进程代码段度量10次所用时间的均值.

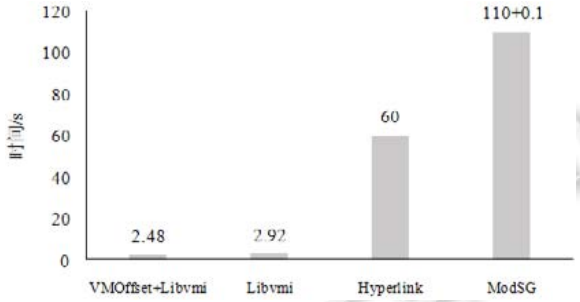


Fig.12 Process view reconstruction time

图12 进程视图重构时间

Table 4 Time for VMOffset measuring the process code segment

表4 VMOffset度量进程代码段时间

进程名称	进程pid	进程位置	代码段页数	度量时间/s
init	1	2	46	1.64
getty	768	53	6	1.89
sshd	455	36	128	2.21
bluetoothd	505	41	210	3.28
sh	1 489	93	24	3.52
xorg	869	63	498	10.10

图12中,VMOffset+Libvmi为修改Libvmi使其直接使用VMOffset所得进程偏移量进行视图重构,第2列的Libvmi则为初始通过读取配置文件中的进程偏移量以重构TVM进程视图.实验时,TVM进程链表中共有97个进程.由图可知:VMOffset+Libvmi重构进程视图所用时间为2.48s,Libvmi为2.92s.可见:前者无需手动配置进程偏移量,且重构进程链表所用时间相比原始Libvmi要少,效率更高.图12还列出了两种具有代表性的VMI方法的进程视图重构时间,其中,HyperLink[27]基于内存搜索完成语义重构,其重构进程链表需 1 分钟左右;ModSG[28]则通过在后端离线解析不同版本内核结构生成语义静态库,前端模块基于语义静态库完成视图的重构.ModSG生成Ubuntu12.04操作系统对应的语义静态库所用时间约为110s,当TVM中进程数量为100个时,其前端重构时间约为0.1s,因此,其重构进程视图时间为110.1s.对比可知,VMOffset重构TVM中进程视图的效率是较高的.

表4中,进程位置用于说明待度量进程为TVM进程链表中的第几个进程,代码段页数为待度量进程代码段总页数.进程代码段的度量需首先遍历进程链表判断待度量进程是否存在,此后获取代码段各页内容进行度量,因此其效率与进程位置及代码段页数两者相关.由表4可知:所度量进程在链表中的位置越靠后,以及代码段页数越大,所需度量时间也越长.多数进程的度量时间在 3.5s 之内,但当进程所处位置较后,且代码段页数较大时,其度量时间也会相应增加,如Xorg进程度量所用时间为10s左右.因代码段的重构与度量均在TVM外部实现,对TVM本身不会产生影响,结合需要进行的哈希运算与地址翻译的次数来看,这样的时间效率是可以接受的.

为了进一步评估VMOffset的有效性,本节将VMOffset与现有部分VMI方法进行了对比,见表5.

表5中,VMwatcher的自省过程依赖内核源码,Virtuoso及VMST需引入与TVM环境一致的SVM,Volatility,Libvmi及vDetector需在TVM中加载内核模块获取进程偏移量,ModSG需在后端离线解析不同版本内核结构生成语义静态库.上述方法对于不同内核版本的TVM,均需要重新修改、配置或解析.其中,HyperLink的实现对

TVM 内核版本不敏感,但其重构进程链表需 1 分钟左右,且仅能重构进程号及进程名称.对比之下,VMOffset 具有内核版本无关性,无需额外引入 SVM 或访问 TVM,其可重构的进程级语义较为丰富,且重构效率高、性能损耗小;此外,VMOffset 所得进程偏移量可作为一个信息段随 TVM 的迁移而迁移,具有较强的通用性与可移植性.

Table 5 Comparison of VMOffset and other VMI methods

表 5 VMOffset 与其他 VMI 方法的对比

方法	内核源码	安全虚拟机	访问目标虚拟机	内核版本敏感	性能&效率
VMwatcher[20]	需要	不需要	不需要	敏感	一般
Virtuoso[14]	不需要	需要	不需要	敏感	较差
VMST[15]	不需要	需要	不需要	敏感	较差
Volatility[9]	不需要	不需要	需要	敏感	较好
Libvmi[10]	不需要	不需要	需要	敏感	较好
vDetector[3]	不需要	不需要	需要	敏感	较好
HyperLink[27]	不需要	不需要	不需要	不敏感	一般
ModSG[28]	不需要	不需要	不需要	敏感	一般
VMOffset	不需要	不需要	不需要	不敏感	较好

4 结 语

本文针对现有虚拟机自省方法中存在的实现过程与 TVM 内核版本相关、移植性差、系统执行效率不高的问题,提出了一种虚拟机自省中的语义重构改进方法 VMOffset.VMOffset 基于 KVM-QEMU 虚拟化平台实现,可在未知 TVM 内核版本的前提下,自动获取 TVM 中进程偏移量并提供给自研或开源 VMI 工具完成语义重构,具有较好的可移植性与安全性.VMOffset 无需修改 TVM,在 TVM 启动阶段完成进程偏移量的获取,执行效率较高,且引入的性能损耗在 0.05%以内.同时,VMOffset 也存在一些局限性,如要求硬件虚拟化技术为 Intel VT-x,且目前仅支持 Linux 操作系统的 TVM.在下一步的工作中,将探索 VMOffset 对 Linux 以外操作系统的支持,并丰富 VMOffset 在重构文件、网络连接等信息上的功能,实现更强的通用性及更细粒度的虚拟机自省.

References:

[1] Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. In: Neuman C, ed. Proc. of the Network and Distributed Systems Security Symp. Washington: Internet Society, 2003. 191−206.

[2] Li BH, Xu KF, Zhang P, Guo L, Hu Y, Fang BX. Research and application progress of virtual machine introspection technology. Ruan Jian Xue Bao/Journal of Software, 2016,27(6):1384−1401 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5006.htm [doi: 10.13328/j.cnki.jos.005006]

[3] Li B, Wo TY, Hu CM, Li JX, Wang Y, Huai JP. Hidden OS objects correlated detection technology based on VMM. Ruan Jian Xue Bao/Journal of Software, 2013,24(2):405−420 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/24/405.htm [doi: cnki:sun:rjxb.0.2013-02-017]

[4] Pfoh J, Schneider C, Eckert C. A formal model for virtual machine introspection. In: Nieh J, Stavrou A, eds. Proc. of the 1st ACM Workshop on Virtual Machine Security. New York: ACM, 2009. 1−10. [doi: 10.1145/1655148.1655150]

[5] Dinaburg A, Royal P, Sharif M, Lee W. Ether: Malware analysis via hardware virtualization extensions. In: Ning P, ed. Proc. of the 15th ACM Conf. on Computer and Communications Security. New York: ACM, 2008. 51−62. [doi: 10.1145/1455770.1455779]

[6] Xiong X, Tian D, Liu P. Practical protection of kernel integrity for commodity OS from untrusted extensions. In: Proc. of the 18th Annual Network and Distributed System Security Symp. Washington: Internet Society, 2011. 1−17.

[7] Jain B, Baig MB, Zhang DL, Porter DE, Sion R. SOK: Introspections on trust and the semantic GAP. In: Proc. of the 2014 IEEE Symp. on Security and Privacy (SP). Washington: IEEE Computer Society, 2014. 605−620. [doi: 10.1109/SP.2014,45]

[8] Vogl S, Kilic F, Schneider C, Eckert C. X-TIER: Kernel module injection. In: Lopez J, Huang XY, Sandhu R, eds. Proc. of the Network and System Security. LNCS 7873, 2013. 192−205. [doi: 10.1007/978-3-642-38631-2_15]

[9] Volatility. Volatile memory analysis framework. https://github.com/volatilityfoundation/volatility

[10] Xiong H, Liu Z, Xu W, Jiao S. Libvmi: A library for bridging the semantic gap between guest OS and VMM. In: Proc. of the 12th IEEE Int'l Conf. on Computer and Information Technology. Washington: IEEE Computer Society, 2012. 549−556. [doi: 10.1109/CIT.2012.119]

[11] Payne BD. Xen access library. https://code.google.com/p/xenaccess/

[12] Jones ST, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Antfarm: Tracking processes in a virtual machine environment. In: Proc. of the USENIX Annual Technical Conf. New York: ACM, 2006. 1−14.

[13] Srinivasan D, Wang Z, Jiang XX, Xu DY. Process out-grafting: An efficient "out-of-VM" approach for fine-grained process execution monitoring. In: Proc. of the 18th ACM Conf. on Computer and Communications Security. New York: ACM, 2011. 363−374. [doi: 10.1145/2046707,2046751]

[14] Dolan-Gavitt B, Leek T, Zhivich M, Giffin J, Lee W. Virtuoso: Narrowing the semantic GAP in virtual machine introspection. In: Proc. of the 32nd IEEE Symp. on Security and Privacy. Washington: IEEE Computer Society, 2011. 297−312. [doi: 10.1109/SP.2011,11]

[15] Fu YC, Lin ZQ. Space traveling across VM: Automatically bridging the semantic GAP in virtual machine introspection via online kernel data redirection. In: Proc. of the IEEE Symp. on Security and Privacy. Washington: IEEE Computer Society, 2012. 586−600. [doi: 10.1109/SP.2012,40]

[16] Saberi A, Fu YC, Lin Z. Hybrid-bridge: Efficiently bridging the semantic GAP in virtual machine introspection via decoupled execution and training memorization. In: Proc. of the 21st Annual Network and Distributed System Security Symp. Washington: Internet Society, 2014. 1−15. [doi: 10.14722/ndss.2014.23226]

[17] Carbone M, Cui WD, Lu L, Lee W, Peinado M, Jiang XX. Mapping kernel objects to enable systematic integrity checking. In: Proc. of the 16th ACM Conf. on Computer and Communications Security (CCS 2009). New York: ACM, 2009. 555−565. [doi: 10.1145/1653662,1653729]

[18] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. Copenhagen: University of Copenhagen, 1994.

[19] Schneider C, Pfoh J, Echert C. Bridging the semantic GAP through static code analysis. In: Proc. of the 2012 European Workshop on System Security (EuroSec 2012). New York: ACM, 2012. 1−6.

[20] Jiang XX, Wang XY, Xu DY. Stealthy Malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In: Ning P, ed. Proc. of the 14th ACM Conf. on Computer and Communications Security. New York: ACM, 2007. 128−138. [doi: 10.1145/1315245,1315262]

[21] Inoue H, Adelstein F, Donovan M, Brueckner S. Automatically bridging the semantic GAP using c interpreter. In: Proc. of the 2011 Annual Symp. on Information Assurance. 2011. 51−58.

[22] Volatilitux: Physical memory analysis of linux systems. http://code.google.com/p/volatilitux

[23] Pfoh J, Schneider C, Eckert C. Nitro: Hardware-based system call tracing for virtual machines. In: Tetsu I, Nishigaki M, eds. Proc. of the 6th Int'l Conf. on Advances in Information and Computer Security. Washington: IEEE Computer Society, 2011. 96−112. [doi: 10.1007/978-3-642-25141-2_7]

[24] Chen XS, Chen MM, Jin X. Shadow memory-based agentless virtual machine process protection. Journal of University of Electronic Science and Technology of China, 2018,47(1):80−87 (in Chinese with English abstract). [doi: cnki:sun:dkdx.0.2018-01-012]

[25] OpenCIT: Open continuous integration and test. http://opencit.openengsb.org

[26] Cai MJ, Chen XS, Jin X, Zhao C, Yin MY. Paging-measurement method for virtual machine process code based on hardware virtualization. Journal of Computer Applications, 2018,38(2):305−309 (in Chinese with English abstract). [doi: 10.11772/j.issn.1001-9081.2017082167]

[27] Xiao JD, Lu L, Wang HN, Zhu XY. HyperLink: Virtual machine introspection and memory forensic analysis without kernel source code. In: Proc. of the IEEE Int'l Conf. on Autonomic Computing. Washington: IEEE Computer Society, 2016. 127−136. [doi: 10.1109/icac.2016.46]

[28] Cui CY, Wu Y, Li P, Zhang XM. Narrowing the semantic GAP in virtual machine introspection. Journal on Communications, 2015, 36(8):31−37 (in Chinese with English abstract). [doi: cnki:sun:txxb.0.2015-08-005]

附中文参考文献:

[2] 李保珲,徐克付,张鹏,郭莉,胡玥,方滨兴.虚拟机自省技术研究与应用进展.软件学报,2016,27(6):1384−1401. http://www.jos.org.cn/1000-9825/5006.htm [doi: 10.13328/j.cnki.jos.005006]

[3] 李博,沃天宇,胡春明,李建欣,王颖,怀进鹏.基于 VMM 的操作系统隐藏对象关联检测技术.软件学报,2013,24(2):405−420. http://www.jos.org.cn/1000-9825/24/405.htm [doi: cnki:sun:rjxb.0.2013-02-017]

[24] 陈兴蜀,陈蒙蒙,金鑫.基于影子内存的无代理虚拟机进程防护.电子科技大学学报,2018,47(1):80−87. [doi: cnki:sun:dkdx.0.2018-01-012]

[26] 蔡梦娟,陈兴蜀,金鑫,等.基于硬件虚拟化的虚拟机进程代码分页式度量方法.计算机应用,2018,38(2):305−309. [doi: 10.11772/j.issn.1001-9081.2017082167]

[28] 崔超远,乌云,李平,等.虚拟机自省中一种消除语义鸿沟的方法.通信学报,2015,36(8):31−37. [doi: cnki:sun:txxb.0.2015-08-005]

陈兴蜀(1968－),女,博士,教授,博士生导师,CCF 专业会员,主要研究领域为云计算/大数据安全,威胁检测,开源情报.

王启旭(1985－),男,博士,助理研究员,主要研究领域为云计算安全,数据隐私保护,物联网安全.

蔡梦娟(1996－),女,硕士,主要研究领域为云计算,虚拟化安全.

金鑫(1976－),男,博士,讲师,主要研究领域为云计算安全,可信计算.

王伟(1992－),男,硕士,CCF 学生会员,主要研究领域为云计算,可信计算,数据库.